State equality detection for implementation-level model-checking of distributed applications

Marion Guthmuller

State equality detection for implementation-level model-checking of distributed applications

Marion Guthmuller[1,2,3]

[1] Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France.
[2] CNRS, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France
[3] Inria, AlGorille project-team, Villers-lès-Nancy, F-54600, France
marion.guthmuller@loria.fr

Abstract. The work presented in this paper occurs in the context of validating implementations of distributed algorithms through model-checking. Among the properties that can be checked, we focus on the verification of liveness properties and the detection of acceptance cycle. We propose a new heap comparison algorithm based on a re-implementation of `malloc`, which reduces the amount of false negatives in the system state equality detection, thanks to system-level information analysis.

1 Introduction

Distributed systems such as grids, peer-to-peer systems or high-performance supercomputers benefit of an ever increasing popularity nowadays. Distributed applications are executed routinely on these systems. However, the design and implementation of this kind of applications remain difficult and often lead to numerous bugs. The classical problems faced in this context are race conditions, livelocks and deadlocks. In addition, the distributed processes do not share a global clock and there is no direct access to a global state of the application. It is thus harder to manage the coherency and validity of the application behavior. Therefore, it is crucial to get a validation step that assesses the validity of a distributed application.

Nevertheless, this correctness verification remains almost impossible to do using simple testing approaches. Indeed, any number of executions, even huge, does not guarantee that the properties are valid in any case. For example, one problem may occur only one time over one billion executions according to a particular context of execution. So, the validation must consider every possible executions, and this can be achieved through formal methods. The relative simplicity of use of the model-checking explains the attention received by this method recently.

The properties that can be checked on any system can be classified as either safety properties or liveness ones. Safety properties enforce that a given bad behavior never occurs while liveness properties ensure that an expected behavior will happen in all cases. Doing nothing easily fulfills a safety property as this will

never lead to a *"bad"* situation. Thus, we need to include a notion of progress in the verification, thanks to liveness properties. Typically, enforcing safety properties is easier than liveness ones, that usually reveal much harder to verify since they depend not only on separate system states, but also on system evolution paths.

Within model-checking, two approaches are possible. The first one works at the model level and makes use of model-checkers while the second one comes from the fact that there is always a gap between an algorithmic model and its possible implementations. Hence, although a model can be theoretically validated, its manual conversion into a practical implementation may still contain errors. That is why the second approach operates directly at the implementation level and aims at validating an actual program. This latter approach constitutes the generic context of this work.

The direct model-checking of software-level applications poses technical difficulties in addition to the classical more theoretical ones. The state space explosion that can be found in any model-checking approach is still to be dealt with, but dealing with a real application is much harder than simulating an abstract model. To explore all relevant execution paths, the model-checker must firmly control the execution of the tested application. It must be able to control which branch is taken at a given point (for example by controlling the order in which messages are dispatched to components), and it must also be able to rewind the application into a previous state to explore another possible branch. Another severe difficulty posed by software-level model-checking is the gathering of qualitative information about the system state that are required by the model-checking algorithms (e.g. for state space reduction). For example, retrieving the value of a given variable is trivial when simulating abstract models, but reveals much more challenging at software-level.

This work presents several tools aiming at filling this gap, so that classical model-checking algorithms become applicable directly at software-level. We rely on the SimGrid [1] simulator of parallel and distributed applications to control the tested application, and propose several new system-level techniques to gather information about the application.

The following section presents a brief state of the art about model and implementation-level checking. Then, our contributions are presented in Section 3 while the last section concludes the paper and presents some envisioned future directions.

2 State of the art

2.1 Abstract model-checking

SPIN : this model-checker [4] is currently the best-known open-source model-checker for the verification of distributed systems. Developed by Bell Labs, this tool allows verification of applications previously modeled with language PROMELA (Process Meta Language) by detecting the presence of deadlocks,

race conditions or unwarranted assumptions about the relative speeds of processes. Correctness properties can be specified using assertions or LTL (Linear Temporal Logic). Moreover, thanks to their modeling language, it performs a static analysis and a partial order reduction to reduce the state space combined with on-the-fly exploration, which means that it avoids the need to build a global state graph.

MaceMC : this project [5], developed within the Mace tool, enables verification of safety and liveness properties for distributed systems written in C++. As SPIN, it uses a specific formalization (Mace) for modeling and verification of these systems. To deal with the explosion of state space, this model-checker uses a characterization of states in categories, living or dead, and the search for so-called critical transitions, that is to say, the moment at which it will be impossible to return to a living state.

2.2 Implementation-level model-checking

Verisoft : it is the first model-checker [3] to enable verification of concurrent systems written in C or C++, without using a prior modeling formalism in a specific model-checker. It also provides, for the first time, a stateless search algorithm that does not store the visited states (limiting the memory consumption) and that exploits the independence of transitions to deal with the combinatorial explosion of state space. However, the project development stopped in 2006 at a point where only the verification of safety properties was available.

CMC (C Model -Checker) : this model-checker [7] works on unmodified C or C++ implementations, targeting subtle bugs in system code. The checked correctness properties are assertion violations, sanity checks on table entries and messages and memory errors. To deal with the complexity of comparing the running state of C process, it uses a canonical representation of the memory. Moreover, unlike Verisoft, it uses a hash table to store a small signature of each visited state. This last technique enables CMC to handle a many orders of magnitude larger number of states but at the risk of missing errors due to hash collisions.

JavaPathFinder : this is a model-checker [8] for Java bytecode developed by NASA. It uses a custom Java Virtual Machine that executes the bytecode instructions with a search component that guides the execution. To deal with the state explosion, this model-checker combines the use of symmetry reductions, abstract interpretation, static analysis and runtime analysis. JavaPathFinder is especially useful to verify concurrent Java programs. It can check for deadlocks, invariants and user-defined assertions in the code, formulated with LTL (Linear Temporal Logic).

SimGridMC : this project [6] uses a dynamic approach for the verification of safety properties on distributed applications written in C, for the SimGrid simulation framework, using any of several communication APIs provided by the simulator. Moreover, it exploits stateless algorithm implemented for Verisoft with a dynamic partial order reduction [2], using the notion of independence between transitions, to reduce state space exploration. It currently enables the verification of local assertions and the absence of deadlocks. The contributions presented in this paper are part of the extension of this model-checker to allow verification of liveness properties.

3 State equality detection

The verification of liveness properties is performed by looking for acceptance cycles in all the possible successive global states of the application. To detect an acceptance cycle in the execution path of the tested application, we need to check if any visited state has already been visited on this path, in the case we have an acceptance state. A state is composed of the system state, defined by the global variables, stack and heap of the OS-process executing the system, and the Büchi automaton state corresponding to the negation of the property, defined by the values of atomic propositions of the LTL formula. The atomic propositions correspond to boolean variables. The comparison of those variables can be done thanks to primitives that must be provided by the user and that return their respective values. The main difficulty relates to the comparison of system state.

The most common way to compare two system states is by performing a simple `memcmp` on the stored memory areas. However, those considered areas contain a lot of information whose details are not important to the application semantic, such as the numerical address returned by `malloc` or the port on which `accept` binded the socket service. That leads to false negatives where the model-checker thinks that two states differ from each other whereas they are equal from the application point of view. To reduce the amount of false negatives, we are working on a notion of state distance instead of perfect equality. For this, we propose a new heap comparison algorithm based on a re-implementation of `malloc`.

3.1 Re-implementation of `malloc`

This re-implementation is based on the `mmalloc` GNU package which enables the splitting of the heap in two parts and choose the memory area wherein the allocation must be done. To explore all the possible executions, when we arrive at the end of an execution path, we need to backtrack to a previous state in order to explore another path. To perform the backtracking, we retrieve the initial global state stored at the beginning of the execution, we restore the process states from this snapshot and then we replay the previously considered execution until it reaches the global state from which we want to continue the

exploration. Thanks to `mmalloc`, the initial state and all transitions explored on a path are stored on a different heap which is not deleted by the backtracking mechanism.

Moreover, we have added two extra meta-data for each allocation performed during the execution. The first one is the requested size when `malloc` is called. The heap is divided into blocks of 4096 bytes (one page) that can themselves be divided into fragments of identical size equal to a power of 2. If the requested size is inferior to the half size of a block, `malloc` tries to allocate a free fragment with a size superior or equal. If there is no free fragment, a new block divided into fragments is created. For instance, if we want an allocation of 24 bytes, `malloc` will allocate a fragment of 48 bytes. In this case, we don't know how `malloc` fills the unused bytes and that may lead to errors with `memcmp`. The second extra meta-data is the full backtrace at which each malloc were done. This way, the state comparison algorithm can give the location at which divergent blocks were allocated, helping the user to diagnose false negatives.

Finally, to simplify the comparison of the heap, we have adapted the data structure giving per-block information by adding a field indicating explicitly if a block or a fragment is free or used.

3.2 Heap comparison algorithm

From these meta-data, we have implemented a heap comparison algorithm. It consists in comparing (with `memcmp`) each block used or busy fragment. Thanks to the requested size field in the data structure giving per-block information, the comparison is done only on actually used bytes. When `memcmp` returns a value different from zero, we compute the Hamming distance between the blocks or fragments. That allows us to focus on bytes which differ and to obtain their offset. Finally, to determine the location at which the block or fragment was allocated, we display the backtrace stored by `mmalloc`.

By analyzing the backtrace provided by `mmalloc` and thanks to the offset computed with the Hamming distance, we have begun to search manually the variables which differ between the two states. In particular, we were interested in cases where the difference may come simply from a difference of pointer. We are currently working on a solution to determine if the memory areas pointed by each different pointer are equal, which would eliminate false differences. For this, two lists are maintained for comparison: a main list containing addresses of blocks or fragments definitely different and another list, called "critical list", containing those for which a second analysis is needed if they correspond to a pointer. Firstly, we need to find the start address of the potential pointer corresponding to the different byte(s) detected by the algorithm. If it is not a pointer, we add it to the main list of different memory areas. If the pointed area corresponds to another memory area in the heap, two cases are possible. Either the memory area corresponds to a block or a fragment already analyzed, or this area has not yet been compared by the algorithm. In the first case, if the pointed area is not stored in the main list of different memory areas, it means that it was equal. Thus the difference detected is not a real difference. In

the second case, we store the address in the "critical list". When the comparison algorithm works on the pointed block/fragment, if the comparison is positive, we can automatically remove the memory area previously stored from the "critical list". If the comparison is negative, we move the memory area that pointed to the compared memory area into the main list. This mechanism can be executed recursively in the case of pointer of pointer.

Finally, in the case that the difference doesn't correspond to a pointer, another tool is currently under development that will automatically identify the variable or the field of a structure corresponding to the memory area. Given a structure and an offset, the goal is to determine exactly what part of the structure corresponds to the different bytes detected by the heap comparison algorithm.

3.3 Preliminary results

To test the verification of liveness properties in SimGridMC, we have chosen a simple distributed application with three processes. One process corresponds to a coordinator and the others are clients. Each client asks the coordinator a critical section. If it is not used, the coordinator grants the critical section to the client, and then the client releases it after its operation and asks again. If the critical section is used, the request is stored in a queue dealt when the critical section is released. We adapted this example so that one of the client never gets the requested critical section. We check the following liveness property to detect this bug: *"Any process that asks the critical section must obtain it"*.

We studied differences detected between states equal from the application point of view thanks to backtrace. Among the differences, a first category corresponds to differences detected in processes stacks. Indeed, in the case of SimGrid, stacks of processes are allocated in the heap, at the beginning of the simulation. In our example, three memory areas (with a size of 32 blocks for each) are allocated in the heap for the three processes. Only the stack of the model-checker is kept in memory out of the heap. This allows us to separate information about our application and the model-checker to compare only stacks of application processes which compose the system state. The study of these differences will require others tools such as libunwind, for example. A second category of differences is related to pointers and represents at least 30% of the detected differences. The objective at short term is to eliminate these potential false differences with the tool presented in the previous part. Finally, the last category of differences within this example corresponds to communications between processes. Those communications are managed in the SimGrid library and thus their information are located in the heap. At the moment, a manual analysis is necessary to determine their relevance in the state comparison.

Depending on the complexity of the studied applications and the type of the verified property, the number of detected differences may be much larger as they could correspond to other kind of information that the ones enumerated above. Thus, we need to continue the study of examples to enable a better categorization

of differences to detect false negatives and ensure the detection of state equality for the verification of liveness properties.

4 Conclusion and Future Work

In this paper have been presented different tools to address system level issues raised by the verification of temporal properties on distributed applications from the real code. Specifically, we focused on the state equality detection used for the verification of liveness properties. It has been shown that the identification of false negatives is necessary to ensure the validity of the verification but it stays a complex operation. Thanks to a new heap comparison algorithm based on a re-implementation of `malloc`, we can evaluate with a higher precision the distance between two states detected as different by the model-checker. We were able to highlight the irrelevance of some differences leading to false negative detections.

Our proposition enables the detection of some false negatives but not all. The comparison algorithm presented in this paper are intended to help understand why there are false negatives when we work directly at implementation-level of the application for the verification. The goal is to understand where the errors come from to reduce their amount end ensure the relevance of the results.

Our objective at short term is to automatize the analysis of differences targeted by the algorithm. In particular, we had to finalize the mechanism in case of a difference of pointers, working on a heap canonicalization, to automatically remove irrelevant differences, without the user intervention. Moreover, the tool giving the field of a structure corresponding to mismatching bytes, according to the definition of the structure and the offset computed with the Hamming distance, should enable to guide the user in detecting false negatives. Another solution to allow the automation of the analysis for the verification would use information extracted with the heap comparison algorithm, to define a kind of list containing memory allocations known to be irrelevant to the verification.

After the implementation of liveness properties verification, we will study and work on reduction techniques. A preliminary study allowed us to highlight the need to develop new techniques for reducing the state space, since existing techniques applied for safety properties are unsuitable for temporal properties.

Finally, this work represents a first step towards the study of legacy distributed applications at software-level through the verification of liveness properties.

References

1. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation (Mar 2008)
2. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL'05 (2005)

3. Godefroid, P.: Model checking for progamming languages using Verisoft. In: 24th ACM Symposium on Principles of Programming Languages (Jan 1997)
4. Holzmann, G.: The model checker spin. Software Engineering, IEEE Transactions on 23 (may 1997)
5. Killian, C., Anderson, J., Braud, R., Jhala, R., Vahdat, A.M.: Mace : language support for building distributed systems. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (2007)
6. Merz, S., Quinson, M., Rosa, C.: Simgrid mc: Verification support for a multi-api simulation platform. In: 31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems (Jun 2011)
7. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: Cmc: A pragmatic approach to model checking real code. In: Proceedings Fifth Symp. Operating Systems Design and Implementation (OSDI 2002) (2002)
8. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: 15th IEEE international conference on Automated software engineering (2000)